Scilab is not naïve

Michael Baudin

February 2009

Abstract

Most of the time, the mathematical formula is directly used in the Scilab source code. But, in many algorithms, some additionnal work is performed, which takes into account the fact that the computer do not process mathematical real values, but performs computations with their floating point representation. The goal of this article is to show that, in many situations, Scilab is not naïve and use algorithms which have been specifically tailored for floating point computers. We analyse in this article the particular case of the quadratic equation, the complex division and the numerical derivatives, and show that one these examples, the naïve algorithm is not sufficiently accurate.

Contents

1 Introduction

Scilab take cares with your numbers. While most mathematic books deals with exact formulas, Scilab uses algorithms which are specifically designed for computers.

As a practical example of the problem considered in this document, consider the following experiments. The following is an example of a Scilab 5.1 session, where we compute 0.1 by two ways.

```
-->format(25)
-->0.1
 ans  =
    0.10000000000000000055511
-->1.0-0.9
 ans  =
    0.09999999999999999777955
```

I guess that for a person who has never heard of these problems, this experiment may be a shock. To get things clearer, let's check that the sinus function is approximated.

```
-->format(25)
-->sin(0.0)
 ans  =
    0.
-->sin(%pi)
 ans  =
    0.00000000000000001224647
```

The difficulty is generated by the fact that, while the mathematics treat with *real* numbers, the computer deals with their *floating point representations*. This is the difference between the *naive*, mathematical, approach, and the *numerical*, floating-point aware, implementation. (The detailed explanations of the previous examples are presented in the appendix of this document.)

In this article, we will show examples of these problems by using the following theoric and experimental approach.

1. First, we will derive the basic theory at the core of a numerical formula.
2. Then we will implement it in Scilab and compare with the result given by the primitive provided by Scilab. As we will see, some particular cases do not work well with our formula, while the Scilab primitive computes a correct result.
3. Then we will analyse the *reasons* of the differences.

When we compute errors, we use the relative error formula

$$e_r = \frac{|x_c - x_e|}{|x_e|}, \qquad x_e \neq 0 \tag{1}$$

where $x_c \in \mathbb{R}$ is the computed value, and $x_e \in \mathbb{R}$ is the expected value, i.e. the mathematically exact result. The relative error is linked with the number of significant digits in the computed value x_c. For example, if the relative error $e_r = 10^{-6}$, then the number of significant digits is 6.

When the expected value is zero, the relative error cannot be computed, and we then use the absolute error instead

$$e_a = |x_c - x_e|. \tag{2}$$

Before getting into the details, it is important to know that real variables in the Scilab language are stored in *double precision* variables. Since Scilab is following the IEEE 754 standard, that means that real variables are stored with 64 bits precision. As we shall see later, this has a strong influence on the results.

2 Quadratic equation

In this section, we detail the computation of the roots of a quadratic polynomial. As we shall see, there is a whole world from the mathematics formulas to the implementation of such computations. In the first part, we briefly report the formulas which allow to compute the real roots of a quadratic equation with real coefficients. We then present the naïve algorithm based on these mathematical formulas. In the second part, we make some experiments in Scilab and compare our naïve algorithm with the *roots* Scilab primitive. In the third part, we analyse why and how floating point numbers must be taken into account when the implementation of such roots is required.

2.1 Theory

We consider the following quadratic equation, with real coefficients $a, b, c \in \mathbb{R}$ [2, 1, 3] :

$$ax^2 + bx + c = 0. \tag{3}$$

The real roots of the quadratic equations are

$$x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}, \tag{4}$$

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \tag{5}$$

with the hypothesis that the discriminant $\Delta = b^2 - 4ac$ is positive.

The naive, simplified, algorithm which computes the roots of the quadratic is presented in figure 1.

$\Delta \leftarrow b^2 - 4ac$
$s \leftarrow \sqrt{\Delta}$
$x_- \leftarrow (-b - s)/(2a)$
$x_+ \leftarrow (-b + s)/(2a)$

Figure 1: Naive algorithm to compute the real roots of a quadratic equation

2.2 Experiments

The following Scilab function is a straitforward implementation of the previous formulas.

```
1 function r=myroots(p)
2   c=coeff(p,0);
3   b=coeff(p,1);
4   a=coeff(p,2);
5   r=zeros(2,1);
6   r(1)=(-b+sqrt(b^2-4*a*c))/(2*a);
7   r(2)=(-b-sqrt(b^2-4*a*c))/(2*a);
8 endfunction
```

The goal of this section is to show that some additionnal work is necessary to compute the roots of the quadratic equation with sufficient accuracy. We will especially pay attention to rounding errors and overflow problems. In this section, we show that the *roots* command of the Scilab language is not *naive*, in the sense that it takes into account for the floating point implementation details that we will see in the next section.

2.2.1 Rounding errors

We analyse the rounding errors which are appearing when the discriminant of the quadratic equation is such that $b^2 \approx 4ac$. We consider the following quadratic equation

$$\epsilon x^2 + (1/\epsilon)x - \epsilon = 0 \tag{6}$$

with $\epsilon = 0.0001 = 10^{-4}$.

The two real solutions of the quadratic equation are

$$x_- = \frac{-1/\epsilon - \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon} \approx -1/\epsilon^2, \tag{7}$$

$$x_+ = \frac{-1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon} \approx \epsilon^2 \tag{8}$$

The following Scilab script shows an example of the computation of the roots of such a polynomial with the *roots* primitive and with a naive implementation. Only the positive root $x_+ \approx \epsilon^2$ is considered in this test (the x_- root is so that $x_- \to -\infty$ in both implementations).

```
1  p=poly([-0.0001 10000.0 0.0001],"x","coeff");
2  e1 = 1e-8;
3  roots1 = myroots(p);
4  r1 = roots1(1);
5  roots2 = roots(p);
6  r2 = roots2(1);
7  error1 = abs(r1-e1)/e1;
8  error2 = abs(r2-e1)/e1;
9  printf("Expected : %e\n", e1);
10 printf("Naive method : %e (error=%e)\n", r1,error1);
11 printf("Scilab method : %e (error=%e)\n", r2, error2);
```

The script then prints out :

```
Expected : 1.000000e-008
Naive method : 9.094947e-009 (error=9.050530e-002)
Scilab method : 1.000000e-008 (error=1.654361e-016)
```

The result is surprising, since the naive root has no correct digit and a relative error which is 14 orders of magnitude greater than the relative error of the Scilab root.

The explanation for such a behaviour is that the expression of the positive root is the following

$$x_+ = \frac{-1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon} \tag{9}$$

and is numerically evalutated as

```
\sqrt{1/\epsilon^2+4\epsilon^2} = 10000.000000000001818989
```

As we see, the first digits are correct, but the last digits are polluted with rounding errors. When the expression $-1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2}$ is evaluated, the following computations are performed :

```
-1/\epsilon+ \sqrt{1/\epsilon^2+4\epsilon^2}
  = -10000.0 + 10000.000000000001818989
  = 0.00000000000018189894035
```

The user may think that the result is extreme, but it is not. Reducing furter the value of ϵ down to $\epsilon = 10^{-11}$, we get the following output :

```
Expected : 1.000000e-022
Naive method : 0.000000e+000 (error=1.000000e+000)
Scilab method : 1.000000e-022 (error=1.175494e-016)
```

The relative error is this time 16 orders of magnitude greater than the relative error of the Scilab root. In fact, the naive implementation computes a false root x_+ even for a value of epsilon equal to $\epsilon = 10^-3$, where the relative error is 7 times greater than the relative error produced by the *roots* primitive.

2.2.2 Overflow

In this section, we analyse the overflow exception which is appearing when the discriminant of the quadratic equation is such that $b^2 >> 4ac$. We consider the following quadratic equation

$$x^2 + (1/\epsilon)x + 1 = 0 \tag{10}$$

with $\epsilon \to 0$.

The roots of this equation are

$$x_- \approx -1/\epsilon \to -\infty, \qquad \epsilon \to 0 \tag{11}$$

$$x_+ \approx -\epsilon \to 0^-, \qquad \epsilon \to 0 \tag{12}$$

To create a difficult case, we search ϵ so that $1/\epsilon^2 = 10^{310}$, because we know that 10^{308} is the maximum value available with double precision floating point numbers. One possible solution is $\epsilon = 10^{-155}$.

The following Scilab script shows an example of the computation of the roots of such a polynomial with the *roots* primitive and with a naive implementation.

```
1  // Test #3 : overflow because of b
2  e=1.e-155
3  a = 1;
4  b = 1/e;
5  c = 1;
6  p=poly([c b a],"x","coeff");
7  expected = [-e;-1/e];
8  roots1 = myroots(p);
9  roots2 = roots(p);
10 error1 = abs(roots1-expected)/norm(expected);
11 error2 = abs(roots2-expected)/norm(expected);
12 printf("Expected : %e %e\n", expected(1),expected(2));
13 printf("Naive method : %e %e (error=%e)\n", roots1(1),roots1(2),error1);
14 printf("Scilab method : %e %e (error=%e)\n", roots2(1),roots2(2), error2);
```

The script then prints out :

```
Expected : -1.000000e-155 -1.000000e+155
Naive method : Inf Inf (error=Nan)
Scilab method : -1.000000e-155 -1.000000e+155 (error=0.000000e+000)
```

As we see, the $b^2 - 4ac$ term has been evaluated as $1/\epsilon^2 - 4$, which is approximately equal to 10^{310}. This number cannot be represented in a floating point number. It therefore produces the IEEE overflow exception and set the result as *Inf*.

2.3 Explanations

The following tricks are extracted from the *quad* routine of the *RPOLY* algorithm by Jenkins [11]. This algorithm is used by Scilab in the roots primitive, where a special case is handled when the degree of the equation is equal to 2, i.e. a quadratic equation.

2.3.1 Properties of the roots

One can easily show that the sum and the product of the roots allow to recover the coefficients of the equation which was solve. One can show that

$$x_- + x_+ = \frac{-b}{a} \tag{13}$$

$$x_- x_+ = \frac{c}{a} \tag{14}$$

Put in another form, one can state that the computed roots are solution of the normalized equation

$$x^2 - \left(\frac{x_- + x_+}{a}\right) x + x_- x_+ = 0 \tag{15}$$

Other transformation leads to an alternative form for the roots. The original quadratic equation can be written as a quadratic equation on $1/x$

$$c(1/x)^2 + b(1/x) + a = 0 \tag{16}$$

Using the previous expressions for the solution of $ax^2+bx+c = 0$ leads to the following expression of the roots of the quadratic equation when the discriminant is positive

$$x_- = \frac{2c}{-b + \sqrt{b^2 - 4ac}}, \tag{17}$$

$$x_+ = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \tag{18}$$

These roots can also be computed from 4, with the multiplication by $-b + \sqrt{b^2 - 4ac}$.

2.3.2 Conditionning of the problem

The conditionning of the problem may be evaluated with the computation of the partial derivatives of the roots of the equations with respect to the coefficients. These partial derivatives measure the sensitivity of the roots of the equation with respect to small errors which might pollute the coefficients of the quadratic equations.

In the following, we note $x_- = \frac{-b-\sqrt{\Delta}}{2a}$ and $x_+ = \frac{-b+\sqrt{\Delta}}{2a}$ when $a \neq 0$. If the discriminant is stricly positive and $a \neq 0$, i.e. if the roots of the quadratic are real, the partial derivatives of the roots are the following :

$$\frac{\partial x_-}{\partial a} = \frac{c}{a\sqrt{\Delta}} + \frac{b+\sqrt{\Delta}}{2a^2}, \qquad a \neq 0, \qquad \Delta \neq 0 \tag{19}$$

$$\frac{\partial x_+}{\partial a} = -\frac{c}{a\sqrt{\Delta}} + \frac{b-\sqrt{\Delta}}{2a^2} \tag{20}$$

$$\frac{\partial x_-}{\partial b} = \frac{-1-b/\sqrt{\Delta}}{2a} \tag{21}$$

$$\frac{\partial x_+}{\partial b} = \frac{-1+b/\sqrt{\Delta}}{2a} \tag{22}$$

$$\frac{\partial x_-}{\partial c} = \frac{1}{\sqrt{\Delta}} \tag{23}$$

$$\frac{\partial x_+}{\partial c} = -\frac{1}{\sqrt{\Delta}} \tag{24}$$

If the discriminant is zero, the partial derivatives of the double real root are the following :

$$\frac{\partial x_\pm}{\partial a} = \frac{b}{2a^2}, \qquad a \neq 0 \tag{25}$$

$$\frac{\partial x_\pm}{\partial b} = \frac{-1}{2a} \tag{26}$$

$$\frac{\partial x_\pm}{\partial c} = 0 \tag{27}$$

The partial derivates indicate that if $a \approx 0$ or $\Delta \approx 0$, the problem is ill-conditionned.

2.3.3 Floating-Point implementation : fixing rounding error

In this section, we show how to compute the roots of a quadratic equation with protection against rounding errors, protection against overflow and a minimum amount of multiplications and divisions.

Few but important references deals with floating point implementations of the roots of a quadratic polynomial. These references include the important paper [9] by Golberg, the Numerical Recipes [18], chapter 5, section 5.6 and [6], [17], [13].

The starting point is the mathematical solution of the quadratic equation, depending on the sign of the discriminant $\Delta = b^2 - 4ac$:

- If $\Delta > 0$, there are two real roots,

$$x_\pm = \frac{-b \pm \sqrt{\Delta}}{2a}, \qquad a \neq 0 \tag{28}$$

- If $\Delta = 0$, there are one double root,

$$x_\pm = -\frac{b}{2a}, \qquad a \neq 0 \tag{29}$$

- If $\Delta < 0$,

$$x_\pm = \frac{-b}{2a} \pm i\frac{\sqrt{-\Delta}}{2a}, \qquad a \neq 0 \tag{30}$$

In the following, we make the hypothesis that $a \neq 0$.

The previous experiments suggest that the floating point implementation must deal with two different problems :

- rounding errors when $b^2 \approx 4ac$ because of the cancelation of the terms which have opposite signs,
- overflow in the computation of the discriminant Δ when b is large in magnitude with respect to a and c.

When $\Delta > 0$, the rounding error problem can be splitted in two cases

- if $b < 0$, then $-b + \sqrt{b^2 - 4ac}$ may suffer of rounding errors,
- if $b > 0$, then $-b - \sqrt{b^2 - 4ac}$ may suffer of rounding errors.

Obviously, the rounding problem will not appear when $\Delta < 0$, since the complex roots do not use the sum $-b + \sqrt{b^2 - 4ac}$. When $\Delta = 0$, the double root does not cause further trouble. The rounding error problem must be solved only when $\Delta > 0$ and the equation has two real roots.

A possible solution may found in combining the following expressions for the roots

$$x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}, \tag{31}$$

$$x_- = \frac{2c}{-b + \sqrt{b^2 - 4ac}}, \tag{32}$$

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \tag{33}$$

$$x_+ = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \tag{34}$$

The trick is to pick the formula so that the sign of b is the same as the sign of the square root. The following choice allow to solve the rounding error problem

- compute x_- : if $b < 0$, then compute x_- from 32, else (if $b > 0$), compute x_- from 31,
- compute x_+ : if $b < 0$, then compute x_+ from 33, else (if $b > 0$), compute x_+ from 34.

The solution of the rounding error problem can be adressed, by considering the modified Fagnano formulas

$$x_1 = -\frac{2c}{b + sgn(b)\sqrt{b^2 - 4ac}}, \tag{35}$$

$$x_2 = -\frac{b + sgn(b)\sqrt{b^2 - 4ac}}{2a}, \tag{36}$$

where

$$sgn(b) = \begin{cases} 1, \text{ if } b \geq 0, \\ -1, \text{ if } b < 0, \end{cases} \tag{37}$$

The roots $x_{1,2}$ correspond to $x_{+,-}$ so that if $b < 0$, $x_1 = x_-$ and if $b > 0$, $x_1 = x_+$. On the other hand, if $b < 0$, $x_2 = x_+$ and if $b > 0$, $x_2 = x_-$.

An additionnal remark is that the division by two (and the multiplication by 2) is exact with floating point numbers so these operations cannot be a source of problem. But it is interesting to use $b/2$, which involves only one division, instead of the three multiplications $2 * c$, $2 * a$ and $4 * a * c$. This leads to the following expressions of the real roots

$$x_- = -\frac{c}{(b/2) + sgn(b)\sqrt{(b/2)^2 - ac}}, \tag{38}$$

$$x_+ = -\frac{(b/2) + sgn(b)\sqrt{(b/2)^2 - ac}}{a}, \tag{39}$$

which can be simplified into

$$b' = b/2 \tag{40}$$

$$h = -\left(b' + sgn(b)\sqrt{b'^2 - ac}\right) \tag{41}$$

$$x_1 = \frac{c}{h}, \tag{42}$$

$$x_2 = \frac{h}{a}, \tag{43}$$

where the discriminant is positive, i.e. $b'^2 - ac > 0$.

One can use the same value $b' = b/2$ with the complex roots in the case where the discriminant is negative, i.e. $b'^2 - ac < 0$:

$$x_1 = -\frac{b'}{a} - i\frac{\sqrt{ac - b'^2}}{a}, \tag{44}$$

$$x_2 = -\frac{b'}{a} + i\frac{\sqrt{ac - b'^2}}{a}, \tag{45}$$

A more robust algorithm, based on the previous analysis is presented in figure 2. By comparing 1 and 2, we can see that the algorithms are different in many points.

2.3.4 Floating-Point implementation : fixing overflow problems

The remaining problem is to compute $b'^2 - ac$ without creating unnecessary overflows.

if $a = 0$ **then**
  if $b = 0$ **then**
    $x_- \leftarrow 0$
    $x_+ \leftarrow 0$
  else
    $x_- \leftarrow -c/b$
    $x_+ \leftarrow 0$
  end if
else if $c = 0$ **then**
  $x_- \leftarrow -b/a$
  $x_+ \leftarrow 0$
else
  $b' \leftarrow b/2$
  $\Delta \leftarrow b'^2 - ac$
  if $\Delta < 0$ **then**
    $s \leftarrow \sqrt{-\Delta}$
    $x_1^R \leftarrow -b'/a$
    $x_1^I \leftarrow -s/a$
    $x_2^R \leftarrow x_-^R$
    $x_2^I \leftarrow -x_1^I$
  else if $\Delta = 0$ **then**
    $x_1 \leftarrow -b'/a$
    $x_2 \leftarrow x_2$
  else
    $s \leftarrow \sqrt{\Delta}$
    if $b > 0$ **then**
      $g = 1$
    else
      $g = -1$
    end if
    $h = -(b' + g * s)$
    $x_1 \leftarrow c/h$
    $x_2 \leftarrow h/a$
  end if
end if

Figure 2: A more robust algorithm to compute the roots of a quadratic equation

Notice that a small improvment has allread been done : if $|b|$ is close to the upper bound 10^{154}, then $|b'|$ may be less difficult to process since $|b'| = |b|/2 < |b|$. One can then compute the square root by using normalization methods, so that the overflow problem can be drastically reduced. The method is based on the fact that the term $b'^2 - ac$ can be evaluted with two equivalent formulas

$$b'^2 - ac = b'^2 [1 - (a/b')(c/b')] \quad (46)$$
$$b'^2 - ac = c [b'(b'/c) - a] \quad (47)$$

- If $|b'| > |c| > 0$, then the expression involving $(1 - (a/b')(c/b'))$ is so that no overflow is possible since $|c/b'| < 1$ and the problem occurs only when b is large in magnitude with respect to a and c.
- If $|c| > |b'| > 0$, then the expression involving $(b'(b'/c) - a)$ should limit the possible overflows since $|b'/c| < 1$.

These normalization tricks are similar to the one used by Smith in the algorithm for the division of complex numbers [21].

2.4 References

The 1966 technical report by G. Forsythe [7] presents the floating point system and the possible large error in using mathematical algorithms blindly. An accurate way of solving a quadratic is outlined. A few general remarks are made about computational mathematics. The 1991 paper by Goldberg [9] is a general presentation of the floating point system and its consequences. It begins with background on floating point representation and rounding errors, continues with a discussion of the IEEE floating point standard and concludes with examples of how computer system builders can better support floating point. The section 1.4, "Cancellation" specificaly consider the computation of the roots of a quadratic equation. One can also consult the experiments performed by Nievergelt in [17].

3 Numerical derivatives

In this section, we detail the computation of the numerical derivative of a given function.

In the first part, we briefly report the first order forward formula, which is based on the Taylor theorem. We then present the naïve algorithm based on these mathematical formulas. In the second part, we make some experiments in Scilab and compare our naïve algorithm with the *derivative* Scilab primitive. In the third part, we analyse why and how floating point numbers must be taken into account when the numerical derivatives are to compute.

3.1 Theory

The basic result is the Taylor formula with one variable [10]

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f''''(x) + \mathcal{O}(h^5) \quad (48)$$

If we write the Taylor formulae of a one variable function $f(x)$

$$f(x+h) \approx f(x) + h\frac{\partial f}{\partial x} + \frac{h^2}{2}f''(x) \quad (49)$$

we get the forward difference which approximates the first derivate at order 1

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} + \frac{h}{2}f''(x) \quad (50)$$

The naive algorithm to compute the numerical derivate of a function of one variable is presented in figure 3.

$f'(x) \leftarrow (f(x+h) - f(x))/h$

Figure 3: Naive algorithm to compute the numerical derivative of a function of one variable

3.2 Experiments

The following Scilab function is a straitforward implementation of the previous algorithm.

```
1 function fp = myfprime(f,x,h)
2   fp = (f(x+h) - f(x))/h;
3 endfunction
```

In our experiments, we will compute the derivatives of the square function $f(x) = x^2$, which is $f'(x) = 2x$. The following Scilab script implements the square function.

```
1 function y = myfunction (x)
2   y = x*x;
3 endfunction
```

The most naïve idea is that the computed relative error is small when the step h is small. Because *small* is not a priori clear, we take $\epsilon \approx 10^{-16}$ in double precision as a good candidate for *small*. In the following script, we compare the computed relative error produced by our naïve method with step $h = \epsilon$ and the *derivative* primitive with default step.

```
x = 1.0;
fpref = derivative(myfunction,x,order=1);
e = abs(fpref-2.0)/2.0;
mprintf("Scilab f''=%e, error=%e\n", fpref,e);
h = 1.e-16;
fp = myfprime(myfunction,x,h);
e = abs(fp-2.0)/2.0;
mprintf("Naive f''=%e, h=%e, error=%e\n", fp,h,e);
```

When executed, the previous script prints out :

```
Scilab f'=2.000000e+000, error=7.450581e-009
Naive f'=0.000000e+000, h=1.000000e-016, error=1.000000e+000
```

Our naïve method seems to be quite inaccurate and has not even 1 significant digit ! The Scilab primitive, instead, has 9 significant digits.

Since our faith is based on the truth of the mathematical theory, some deeper experiments must be performed. We then make the following experiment, by taking an initial step $h = 1.0$ and then dividing h by 10 at each step of a loop with 20 iterations.

```
x = 1.0;
fpref = derivative(myfunction,x,order=1);
e = abs(fpref-2.0)/2.0;
mprintf("Scilab f''=%e, error=%e\n", fpref,e);
h = 1.0;
for i=1:20
  h=h/10.0;
  fp = myfprime(myfunction,x,h);
  e = abs(fp-2.0)/2.0;
  mprintf("Naive f''=%e, h=%e, error=%e\n", fp,h,e);
end
```

Scilab then produces the following output.

```
Scilab f'=2.000000e+000, error=7.450581e-009
Naive f'=2.100000e+000, h=1.000000e-001, error=5.000000e-002
Naive f'=2.010000e+000, h=1.000000e-002, error=5.000000e-003
Naive f'=2.001000e+000, h=1.000000e-003, error=5.000000e-004
Naive f'=2.000100e+000, h=1.000000e-004, error=5.000000e-005
Naive f'=2.000010e+000, h=1.000000e-005, error=5.000007e-006
Naive f'=2.000001e+000, h=1.000000e-006, error=4.999622e-007
Naive f'=2.000000e+000, h=1.000000e-007, error=5.054390e-008
Naive f'=2.000000e+000, h=1.000000e-008, error=6.077471e-009
Naive f'=2.000000e+000, h=1.000000e-009, error=8.274037e-008
Naive f'=2.000000e+000, h=1.000000e-010, error=8.274037e-008
Naive f'=2.000000e+000, h=1.000000e-011, error=8.274037e-008
Naive f'=2.000178e+000, h=1.000000e-012, error=8.890058e-005
Naive f'=1.998401e+000, h=1.000000e-013, error=7.992778e-004
Naive f'=1.998401e+000, h=1.000000e-014, error=7.992778e-004
Naive f'=2.220446e+000, h=1.000000e-015, error=1.102230e-001
Naive f'=0.000000e+000, h=1.000000e-016, error=1.000000e+000
Naive f'=0.000000e+000, h=1.000000e-017, error=1.000000e+000
```

```
Naive f'=0.000000e+000, h=1.000000e-018, error=1.000000e+000
Naive f'=0.000000e+000, h=1.000000e-019, error=1.000000e+000
Naive f'=0.000000e+000, h=1.000000e-020, error=1.000000e+000
```

We see that the relative error begins by decreasing, and then is increasing. Obviously, the optimum step is approximately $h = 10^{-8}$, where the relative error is approximately $e_r = 6.10^{-9}$. We should not be surprised to see that Scilab has computed a derivative which is near the optimum.

3.3 Explanations

3.3.1 Floating point implementation

With a floating point computer, the total error that we get from the forward difference approximation is (skipping the multiplication constants) the sum of the linearization error $E_l = h$ (i.e. the $\mathcal{O}(h)$ term) and the rounding error $rf(x)$ on the difference $f(x+h) - f(x)$

$$E = \frac{rf(x)}{h} + \frac{h}{2}f''(x) \tag{51}$$

When $h \to \infty$, the error is then the sum of a term which converges toward $+\infty$ and a term which converges toward 0. The total error is minimized when both terms are equal. With a single precision computation, the rounding error is $r = 10^{-7}$ and with a double precision computation, the rounding error is $r = 10^{-16}$. We make here the assumption that the values $f(x)$ and $f''(x)$ are near 1 so that the error can be written

$$E = \frac{r}{h} + h \tag{52}$$

We want to compute the step h from the rounding error r with a step satisfying

$$h = r^{\alpha} \tag{53}$$

for some $\alpha > 0$. The total error is therefore

$$E = r^{1-\alpha} + r^{\alpha} \tag{54}$$

The total error is minimized when both terms are equal, that is, when the exponents are equal $1 - \alpha = \alpha$ which leads to

$$\alpha = \frac{1}{2} \tag{55}$$

We conclude that the step which minimizes the error is

$$h = r^{1/2} \tag{56}$$

and the associated error is

$$E = 2r^{1/2} \tag{57}$$

Typical values with single precision are $h = 10^{-4}$ and $E = 2.10^{-4}$ and with double precision $h = 10^{-8}$ and $E = 2.10^{-8}$. These are the minimum error which are achievable with a forward difference numerical derivate.

To get a significant value of the step h, the step is computed with respect to the point where the derivate is to compute

$$h = r^{1/2}x \tag{58}$$

One can generalize the previous computation with the assumption that the scaling parameter from the Taylor expansion is h^{α_1} and the order of the formula is $\mathcal{O}(h^{\alpha_2})$. The total error is then

$$E = \frac{r}{h^{\alpha_1}} + h^{\alpha_2} \tag{59}$$

The optimal step is then

$$h = r^{\frac{1}{\alpha_1+\alpha_2}} \tag{60}$$

and the associated error is

$$E = 2r^{\frac{\alpha_2}{\alpha_1+\alpha_2}} \tag{61}$$

An additional trick [18] is to compute the step h so that the rounding error for the sum $x + h$ is minimum. This is performed by the following algorithm, which implies a temporary variable t

$$t = x + h \tag{62}$$
$$h = t - h \tag{63}$$

3.3.2 Results

In the following results, the variable x is either a scalar $x^i n\mathbb{R}$ or a vector $x \in \mathbb{R}^n$. When x is a vector, the step h_i is defined by

$$h_i = (0, \ldots, 0, 1, 0, \ldots, 0) \tag{64}$$

so that the only non-zero component of h_i is the i-th component.

- First derivate : forward 2 points

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \tag{65}$$

  Optimal step : $h = r^{1/2}$ and error $E = 2r^{1/2}$.
  Single precision : $h \approx 10^{-4}$ and $E \approx 10^{-4}$.
  Double precision $h \approx 10^{-8}$ and $E \approx 10^{-8}$.

- First derivate : backward 2 points

$$f'(x) \approx \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h) \tag{66}$$

  Optimal step : $h = r^{1/2}$ and error $E = 2r^{1/2}$.
  Single precision : $h \approx 10^{-4}$ and $E \approx 10^{-4}$.
  Double precision $h \approx 10^{-8}$ and $E \approx 10^{-8}$.

- First derivate : centered 2 points

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} + \mathcal{O}(h^2) \quad (67)$$

Optimal step : $h = r^{1/3}$ and error $E = 2r^{2/3}$.
Single precision : $h \approx 10^{-3}$ and $E \approx 10^{-5}$.
Double precision $h \approx 10^{-5}$ and $E \approx 10^{-10}$.

3.3.3 Robust algorithm

The robust algorithm to compute the numerical derivate of a function of one variable is presented in figure 4.

$h \leftarrow \sqrt{\epsilon}$
$f'(x) \leftarrow (f(x+h) - f(x))/h$

Figure 4: A more robust algorithm to compute the numerical derivative of a function of one variable

3.4 One more step

In this section, we analyse the behaviour of *derivative* when the point x is either large $x \to \infty$, when x is small $x \to 0$ and when $x = 0$. We compare these results with the *numdiff* command, which does not use the same step strategy. As we are going to see, both commands performs the same when x is near 1, but performs very differently when x is large or small.

We have allready explained the theory of the floating point implementation of the *derivative* command. Is it completely *bulletproof* ? Not exactly.

See for example the following Scilab session, where one computes the numerical derivative of $f(x) = x^2$ for $x = 10^{-100}$. The expected result is $f'(x) = 2. \times 10^{-100}$.

```
-->fp = derivative(myfunction,1.e-100,order=1)
 fp  =
    0.000000014901161193847 7
-->fe=2.e-100
 fe  =
    2.000000000000000040-100
-->e = abs(fp-fe)/fe
 e  =
    7.450580596923828243D+91
```

The result does not have any significant digits.

The explanation is that the step is computed with $h = \sqrt{eps} \approx 10^{-8}$. Then $f(x+h) = f(10^{-100} + 10^{-8}) \approx f(10^{-8}) = 10^{-16}$, because the term 10^{-100} is much smaller than 10^{-8}. The result of the computation is therefore $(f(x+h) - f(x))/h = (10^{-16} + 10^{-200})/10^{-8} \approx 10^{-8}$.

The additionnal experiment

```
-->sqrt(%eps)
 ans  =
    0.00000001490116119384 77
```

allows to check that the result of the computation simply is $\sqrt{eps}$. That experiment shows that the *derivative* command uses a wrong defaut step h when x is very small.

To improve the accuracy of the computation, one can take control of the step h. A reasonable solution is to use $h = \sqrt{\epsilon}|x|$ so that the step is scaled depending on x. The following script illustrates than method, which produces results with 8 significant digits.

```
-->fp = derivative(myfunction,1.e-100,order=1,h=sqrt(%eps)*1.e-100)
 fp  =
    2.000000013099139394-100
-->fe=2.e-100
 fe  =
    2.000000000000000040-100
-->e = abs(fp-fe)/fe
 e  =
    0.000000006549569677 0794
```

But when x is exactly zero, the scaling method cannot work, because it would produce the step $h = 0$, and therefore a division by zero exception. In that case, the default step provides a good accuracy.

Another command is available in Scilab to compute the numerical derivatives of a given function, that is *numdiff*. The *numdiff* command uses the step

$$h = \sqrt{\epsilon}(1 + 10^{-3}|x|). \tag{68}$$

In the following paragraphs, we try to analyse why this formula has been chosen. As we are going to check experimentally, this step formula performs better than *derivative* when x is large.

As we can see the following session, the behaviour is approximately the same when the value of x is 1.

```
-->fp = numdiff(myfunction,1.0)
 fp  =
    2.000000018935341739 0237
-->fe=2.0
 fe  =
    2.
-->e = abs(fp-fe)/fe
 e  =
    9.468D-09
```

The accuracy is slightly decreased with respect to the optimal value 7.450581e-009 which was produced by derivative. But the number of significant digits is approximately the same, i.e. 9 digits.

The goal of this step is to produce good accuracy when the value of x is large, where the *numdiff* command produces accurate results, while *derivative* performs poorly.

```
-->numdiff(myfunction,1.e10)
  ans  =
    2.000D+10
-->derivative(myfunction,1.e10,order=1)
 ans  =
    0.
```

This step is a trade-off because it allows to keep a good accuracy with large values of x, but produces a slightly sub-optimal step size when x is near 1. The behaviour near zero is the same, i.e. both commands produce wrong results when $x \to 0$ and $x \neq 0$.

3.5 References

A reference for numerical derivates is [4], chapter 25. "Numerical Interpolation, Differentiation and Integration" (p. 875). The webpage [20] and the book [18] give results about the rounding errors.

4 Complex division

In that section, we analyse the problem of the complex division in Scilab. We especially detail the difference between the mathematical, straitforward formula and the floating point implementation. In the first part, we briefly report the formulas which allow to compute the real and imaginary parts of the division of two complex numbers. We then present the naïve algorithm based on these mathematical formulas. In the second part, we make some experiments in Scilab and compare our naïve algorithm with the / Scilab operator. In the third part, we analyse why and how floating point numbers must be taken into account when the implementation of such division is required.

4.1 Theory

The formula which allows to compute the real and imaginary parts of the division of two complex numbers is

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2} \tag{69}$$

The naive algorithm for the computation of the complex division is presented in figure 5.

$den \leftarrow c^2 + d^2$
$e \leftarrow (ac + bd)/den$
$f \leftarrow (bc - ad)/den$

Figure 5: Naive algorithm to compute the complex division

4.2 Experiments

The following Scilab function is a straitforward implementation of the previous formulas.

```
1 //
2 // naive --
3 //   Compute the complex division with a naive method.
4 //
5 function [cr,ci] = naive (ar , ai , br , bi )
6   den = br * br + bi * bi;
7   cr = (ar * br + ai * bi) / den;
8   ci = (ai * br - ar * bi) / den;
9 endfunction
```

In the following script, one compares the naive implementation against the Scilab implementation with two cases.

```
1   // Check that no obvious bug is in mathematical formula.
2   [cr ci] = naive ( 1.0 , 2.0 , 3.0 , 4.0 )
3   (1.0 + 2.0 * %i)/(3.0 + 4.0 * %i)
4   // Check that mathematical formula does not perform well
5   // when large number are used.
6   [cr ci] = naive ( 1.0 , 1.0 , 1.0 , 1.e307 )
7   (1.0 + 1.0 * %i)/(1.0 + 1.e307 * %i)
```

That prints out the following messages.

```
-->  // Check that no obvious bug is in mathematical formula.
-->  [cr ci] = naive ( 1.0 , 2.0 , 3.0 , 4.0 )
 ci  =
    0.08
 cr  =
    0.44
-->  (1.0 + 2.0 * %i)/(3.0 + 4.0 * %i)
 ans  =
    0.44 + 0.08i
-->  // Check that mathematical formula does not perform well
-->  // when large number are used.
-->  [cr ci] = naive ( 1.0 , 1.0 , 1.0 , 1.e307 )
 ci  =
    0.
 cr  =
    0.
-->  (1.0 + 1.0 * %i)/(1.0 + 1.e307 * %i)
 ans  =
    1.000-307 - 1.000-307i
```

The simple calculation confirms that there is no bug in the naive implementation. But differences are appearing when large numbers are used. In the second case, the naive implementation does not give a single exact digit !

To make more complete tests, the following script allows to compare the results of the naive and the Scilab methods. We use three kinds of relative errors

1. the relative error on the complex numbers, as a whole $e = \frac{|e-c|}{|e|}$,
2. the relative error on the real part $e = \frac{|e_r-e_r|}{e_r}$,
3. the relative error on the imaginary part $e = \frac{|e_i-e_i|}{e_i}$.

```
1  //
2  // compare --
3  //   Compare 3 methods for complex division:
4  //   * naive method
5  //   * Smith method
6  //   * C99 method
7  //
8  function compare (ar, ai, br, bi, rr, ri)
9    printf("****************\n");
10   printf("          a = %10.5e + %10.5e * I\n" , ar , ai );
11   printf("          b = %10.5e + %10.5e * I\n" , br , bi );
12   [cr ci] = naive ( ar, ai, br, bi);
13   printf("Naive --> c = %10.5e + %10.5e * I\n" , cr , ci );
14   c = cr + %i * ci
15   r = rr + %i * ri;
16   error1 = abs(r - c)/abs(r);
17   if (rr==0.0) then
18     error2 = abs(rr - cr);
```

```
19    else
20      error2 = abs(rr - cr)/abs(rr);
21    end
22    if (ri==0.0) then
23      error3 = abs(ri - ci);
24    else
25      error3 = abs(ri - ci)/abs(ri);
26    end
27    printf("   e1 = %10.5e, e2 = %10.5e, e3 = %10.5e\n", error1, error2, error3);
28
29    a = ar + ai * %i;
30    b = br + bi * %i;
31    c = a/b;
32    cr = real(c);
33    ci = imag(c);
34    printf("Scilab ---> c = %10.5e + %10.5e * I\n" , cr ,  ci );
35    c = cr + %i * ci
36    error1 = abs(r - c)/abs(r);
37    if (rr==0.0) then
38      error2 = abs(rr - cr);
39    else
40      error2 = abs(rr - cr)/abs(rr);
41    end
42    if (ri==0.0) then
43      error3 = abs(ri - ci);
44    else
45      error3 = abs(ri - ci)/abs(ri);
46    end
47    printf("   e1 = %10.5e, e2 = %10.5e, e3 = %10.5e\n", error1, error2, error3);
48  endfunction
```

In the following script, we compare the naive and the Scilab implementations of the complex division with 4 couples of complex numbers. The first instruction "ieee(2)" configures the IEEE system so that Inf and Nan numbers are generated instead of Scilab error messages.

```
1  ieee(2);
2  // Check that naive implementation does not have a bug
3  ar = 1;
4  ai = 2;
5  br = 3;
6  bi = 4;
7  rr = 11/25;
8  ri = 2/25;
9  compare (ar, ai, br, bi, rr, ri);
10
11 // Check that naive implementation is not robust with respect to overflow
12 ar = 1;
13 ai = 1;
14 br = 1;
15 bi = 1e307;
16 rr = 1e-307;
17 ri = -1e-307;
18 compare (ar, ai, br, bi, rr, ri);
19
20 // Check that naive implementation is not robust with respect to underflow
21 ar = 1;
```

```
22 ai = 1;
23 br = 1e-308;
24 bi = 1e-308;
25 rr = 1e308;
26 ri = 0.0;
27 compare (ar, ai, br, bi, rr, ri);
```

The script then prints out the following messages.

```
****************
           a = 1.00000e+000 +  2.00000e+000 * I
           b = 3.00000e+000 +  4.00000e+000 * I
Naive  --> c = 4.40000e-001 +  8.00000e-002 * I
   e1 = 0.00000e+000, e2 = 0.00000e+000, e3 = 0.00000e+000
Scilab --> c = 4.40000e-001 +  8.00000e-002 * I
   e1 = 0.00000e+000, e2 = 0.00000e+000, e3 = 0.00000e+000
****************
           a = 1.00000e+000 +  1.00000e+000 * I
           b = 1.00000e+000 +  1.00000e+307 * I
Naive  --> c = 0.00000e+000 +  -0.00000e+000 * I
   e1 = 1.00000e+000, e2 = 1.00000e+000, e3 = 1.00000e+000
Scilab --> c = 1.00000e-307 +  -1.00000e-307 * I
   e1 = 2.09614e-016, e2 = 1.97626e-016, e3 = 1.97626e-016
****************
           a = 1.00000e+000 +  1.00000e+000 * I
           b = 1.00000e-308 +  1.00000e-308 * I
Naive  --> c = Inf +  Nan * I
   e1 = Nan, e2 = Inf, e3 = Nan
Scilab --> c = 1.00000e+308 +  0.00000e+000 * I
   e1 = 0.00000e+000, e2 = 0.00000e+000, e3 = 0.00000e+000
```

The case #2 and #3 shows very surprising results. With case #2, the relative errors shows that the naive implementation does not give any correct digits. In case #3, the naive implementation produces Nan and Inf results. In both cases, the Scilab command "/" gives accurate results, i.e., with at least 16 significant digits.

4.3 Explanations

In this section, we analyse the reason why the naive implementation of the complex division leads to unaccurate results. In the first section, we perform algebraic computations and shows the problems of the naive formulas. In the second section, we present the Smith's method.

4.3.1 Algebraic computations

Let's analyse the second test and check the division of test #2 :

$$\frac{1+I}{1+10^{307}I} = 10^{307} - I * 10^{-307} \tag{70}$$

The naive formulas leads to the following results.

$$
\begin{aligned}
den &= c^2 + d^2 = 1^2 + (10^{307})^2 = 1 + 10^{614} \approx 10^{614} && (71)\\
e &= (ac + bd)/den = (1 * 1 + 1 * 10^{307})/1e614 \approx 10^{307}/10^{614} \approx 10^{-307} && (72)\\
f &= (bc - ad)/den = (1 * 1 - 1 * 10^{307})/1e614 \approx -10^{307}/10^{614} \approx -10^{-307} && (73)
\end{aligned}
$$

To understand what happens with the naive implementation, one should focus on the intermediate numbers. If one uses the naive formula with double precision numbers, then

$$den = c^2 + d^2 = 1^2 + (10^{307})^2 = Inf \tag{74}$$

This generates an overflow, because $(10^{307})^2 = 10^{614}$ is not representable as a double precision number.

The e and f terms are then computed as

$$
\begin{aligned}
e = (ac + bd)/den = (1 * 1 + 1 * 10^{307})/Inf = 10^{307}/Inf = 0 && (75)\\
f = (bc - ad)/den = (1 * 1 - 1 * 10^{307})/Inf = -10^{307}/Inf = 0 && (76)
\end{aligned}
$$

The result is then computed without any single correct digit, even though the initial numbers are all representable as double precision numbers.

Let us check that the case #3 is associated with an underflow. We want to compute the following complex division :

$$\frac{1 + I}{10^{-308} + 10^{-308} I} = 10^{308} \tag{77}$$

The naive mathematical formula gives

$$
\begin{aligned}
den &= c^2 + d^2 = (10^{-308})^2 + (10^{-308})^2 = 10^{-616}10^{-616} + 10^{-616} = 2 \times 10^{-616} && (78)\\
e &= (ac + bd)/den = (1 * 10^{-308} + 1 * 10^{-308})/(2 \times 10^{-616}) && (79)\\
&\approx (2 \times 10^{-308})/(2 \times 10^{-616}) \approx 10^{-308} && (80)\\
f &= (bc - ad)/den = (1 * 10^{-308} - 1 * 10^{-308})/(2 \times 10^{-616}) \approx 0/10^{-616} \approx 0 && (81)
\end{aligned}
$$

With double precision numbers, the computation is not performed this way. Terms which are lower than 10^{-308} are too small to be representable in double precision and will be reduced to 0 so that an underflow occurs.

$$
\begin{aligned}
den &= c^2 + d^2 = (10^{-308})^2 + (10^{-308})^2 = 10^{-616} + 10^{-616} = 0 && (82)\\
e &= (ac + bd)/den = (1 * 10^{-308} + 1 * 10^{-308})/0 \approx 2 \times 10^{-308}/0 \approx Inf && (83)\\
f &= (bc - ad)/den = (1 * 10^{-308} - 1 * 10^{-308})/0 \approx 0/0 \approx NaN && (84)\\
& && (85)
\end{aligned}
$$

4.3.2 The Smith's method

In this section, we analyse the Smith's method and present the detailed steps of this algorithm in the cases #2 and #3.

In Scilab, the algorithm which allows to perform the complex division is done by the the *wwdiv* routine, which implements the Smith's method [22], [9]. The Smith's algorithm is based on normalization, which allow to perform the division even if the terms are large.

The starting point of the method is the mathematical definition

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2} \tag{86}$$

The method of Smith is based on the rewriting of this formula in two different, but mathematically equivalent formulas. The basic trick is to make the terms d/c or c/d appear in the formulas. When c is larger than d, the formula involving d/c is used. Instead, when d is larger than c, the formula involving c/d is used. That way, the intermediate terms in the computations rarely exceeds the overflow limits.

Indeed, the complex division formula can be written as

$$\frac{a+ib}{c+id} = \frac{a+b(d/c)}{c+d(d/c)} + i\frac{b-a(d/c)}{c+d(d/c)} \tag{87}$$

$$\frac{a+ib}{c+id} = \frac{a(c/d)+b}{c(d/c)+d} + i\frac{b(c/d)-a}{c(d/c)+d} \tag{88}$$

These formulas can be simplified as

$$\frac{a+ib}{c+id} = \frac{a+br}{c+dr} + i\frac{b-ar}{c+dr}, \qquad r = d/c \tag{89}$$

$$\frac{a+ib}{c+id} = \frac{ar+b}{cr+d} + i\frac{br-a}{cr+d}, \qquad r = c/d \tag{90}$$

The Smith's method is based on the following algorithm.

if $(|d| <= |c|)$ **then**
  $r \leftarrow d/c$
  $den \leftarrow c + r * d$
  $e \leftarrow (a + b * r)/den$
  $f \leftarrow (b - a * r)/den$
else
  $r \leftarrow c/d$
  $den \leftarrow d + r * c$
  $e \leftarrow (a * r + b)/den$
  $f \leftarrow (b * r - a)/den$
end if

As we are going to check immediately, the Smith's method performs very well in cases #2 and #3.

In the case #2 $\frac{1+i}{1+10^{-308}i}$, the Smith's method is

```
If ( |1e308| <= |1| ) > test false
Else
  r = 1 / 1e308 = 0
  den  = 1e308 +  0 * 1 = 1e308
  e = (1 * 0 + 1) / 1e308 =
  f = (1 * 0 - 1) / 1e308 = -1e-308
```

In the case #3 $\frac{1+i}{10^{-308}+10^{-308}i}$, the Smith's method is

```
If ( |1e-308| <= |1e-308| ) > test true
  r = 1e-308 / 1e308 = 1
  den  = 1e-308 +  1 * 1e-308 = 2e308
  e = (1 + 1 * 1) / 2e308 = 1e308
  f = (1 - 1 * 1) / 2e308 = 0
```

4.4 One more step

In that section, we show the limitations of the Smith's method.

Suppose that we want to perform the following division

$$\frac{10^{307}+i10^{-307}}{10^{2}05+i10^{-205}} = 10^{1}02 - i10^{-}308 \tag{91}$$

The following Scilab script allows to compare the naive implementation and Scilab's implementation based on Smith's method.

```
1 // Check that Smith method is not robust in complicated cases
2 ar = 1e307;
3 ai = 1e-307;
4 br = 1e205;
5 bi = 1e-205;
6 rr = 1e102;
7 ri = -1e-308;
8 compare (ar, ai, br, bi, rr, ri);
```

When executed, the script produces the following output.

```
****************
           a = 1.00000e+307 +  1.00000e-307 * I
           b = 1.00000e+205 +  1.00000e-205 * I
Naive  --> c = Nan +  -0.00000e+000 * I
   e1 = 0.00000e+000, e2 = Nan, e3 = 1.00000e+000
Scilab --> c = 1.00000e+102 +  0.00000e+000 * I
   e1 = 0.00000e+000, e2 = 0.00000e+000, e3 = 1.00000e+000
```

As expected, the naive method produces a Nan. More surprisingly, the Scilab output is also quite approximated. More specifically, the imaginary part is computed as zero, although we know that the exact result is $10^{-}308$, which is representable as a double precision number. The relative error based on the norm of the complex number is accurate ($e1 = 0.0$), but the relative error based on the imaginary part only is wrong ($e3 = 1.0$), without any correct digits.

The reference [5] cites an analysis by Hough which gives a bound for the relative error produced by the Smith's method

$$|zcomp - zref| <= eps|zref| \tag{92}$$

The paper [23] (1985), though, makes a distinction between the norm $|zcomp - zref|$ and the relative error for the real and imaginary parts. It especially gives an example where the imaginary part is wrong.

In the following paragraphs, we detail the derivation of an example inspired by [23], but which shows the problem with double precision numbers (the example in [23] is based on an abstract machine with exponent range ± 99).

Suppose that m, n are integers so that the following conditions are satisfied

$$m >> 0 \tag{93}$$
$$n >> 0 \tag{94}$$
$$n >> m \tag{95}$$

One can easily proove that the complex division can be approximated as

$$\frac{10^n + i10^{-n}}{10^m + i10^{-m}} = \frac{10^{n+m} + 10^{-(m+n)}}{10^{2m} + 10^{-2m}} + i\frac{10^{m-n} - 10^{n-m}}{10^{2m} + 10^{-2m}} \tag{96}$$

Because of the above assumptions, that leads to the following approximation

$$\frac{10^n + i10^{-n}}{10^m + i10^{-m}} \approx 10^{n-m} - i10^{n-3m} \tag{97}$$

which is correct up to approximately several 100 digits.

One then consider $m, n < 308$ but so that

$$n - 3m = -308 \tag{98}$$

For example, the couple $m = 205$, $n = 307$ satisfies all conditions. That leads to the complex division

$$\frac{10^{307} + i10^{-307}}{10^{205} + i10^{-205}} = 10^{102} - i10^{-308} \tag{99}$$

It is easy to check that the naive implementation does not proove to be accurate on that example. We have already shown that the Smith's method is failing to produce a non zero imaginary part. Indeed, the steps of the Smith algorithm are the following

```
If ( |1e-205| <= |1e205| ) > test true
  r = 1e-205 / 1e205 = 0
  den  = 1e205 +  0 * 1e-205 = 1e205
  e = (10^307 + 10^-307 * 0) / 1e205 = 1e102
  f = (10^-307 - 10^307 * 0) / 1e205 = 0
```

The real part is accurate, but the imaginary part has no correct digit. One can also check that the inequality $|zcomp - zref| <= eps|zref|$ is still true.

The limits of Smith's method have been reduced in Stewart's paper [23]. The new algorithm is based on the theorem which states that if $x_1 \dots x_n$ are n floating point representable numbers then $\min_{i=1,n}(x_i) \max_{i=1,n}(x_i)$ is also representable. The algorithm uses that theorem to perform a correct computation.

Stewart's algorithm is superseded by the one by Li et Al [15], but also by Kahan's [12], which, from [19], is the one implemented in the C99 standard.

4.5 References

The 1962 paper by R. Smith [22] describes the algorithm which is used in Scilab. The Goldberg paper [9] introduces many of the subjects presented in this document, including the problem of the complex division. The 1985 paper by Stewart [23] gives insight to distinguish between the relative error of the complex numbers and the relative error made on real and imaginary parts. It also gives an algorithm based on min and max functions. Knuth's bible [14] presents the Smith's method in section 4.2.1, as exercize 16. Knuth gives also references [25] and [8]. The 1967 paper by Friedland [8] describes two algorithm to compute the absolute value of a complex number $|x + iy| = \sqrt{x^2 + y^2}$ and the square root of a complex number $\sqrt{x + iy}$.

5 Conclusion

We have presented several cases where the mathematically perfect algorithm (i.e. without obvious bugs) do not produce accurate results with the computer in particular situations. In this paper, we have shown that specific methods can be used to cure some of the problems. We have also shown that these methods do not cure all the problems.

All Scilab algorithms take floating point values as inputs, and returns floating point values as output. Problems arrive when the intermediate calculations involve terms which are not representable as floating point values.

That article should not discourage us from implementing our own algorithms. Rather, it should warn us and that some specific work is to do when we translate the mathematical material into a algorithm. That article shows us that accurate can be obtained with floating point numbers, provided that we are less *naïve*.

A Simple experiments

In this section, we analyse the examples given in the introduction of this article.

A.1 Why 0.1 is rounded

In this section, we present a brief explanation for the following Scilab session.

```
-->format(25)
-->x1=0.1
 x1  =
    0.10000000000000000055511
-->x2 = 1.0-0.9
 x2  =
    0.09999999999999999777955
-->x1==x2
 ans  =
  F
```

In fact, only the 17 first digits 0.10000000000000005 are significant and the last digits are a artifact of Scilab's displaying system.

The number 0.1 can be represented as the normalized number 1.0×10^{-1}. But the binary floating point representation of 0.1 is approximately [9] $1.1001100110011001100110011001... \times 2^{-4}$. As you see, the decimal representation is made of a finite number of digits while the binary representation is made of an infinite sequence of digits. Because Scilab computations are based on double precision numbers and because that numbers only have 64 bits to represent the number, some *rounding* must be performed.

In our example, it happens that 0.1 falls between two different binary floating point numbers. After rounding, the binary floating point number is associated with the decimal representation "0.10000000000000005", that is "rounding up" in this case. On the other side, 0.9 is also not representable as an exact binary floating point number (but 1.0 is exactly represented). It happens that, after the substraction "1.0-0.9", the decimal representation of the result is "0.09999999999999997", which is different from the rounded value of 0.1.

A.2 Why $sin(\pi)$ is rounded

In this section, we present a brief explanation of the following Scilab 5.1 session, where the function sinus is applied to the number π.

```
-->format(25)
-->sin(0.0)
 ans  =
    0.
-->sin(%pi)
 ans  =
    0.00000000000000001224647
```

Two kinds of approximations are associated with the previous result

- $\pi = 3.1415926...$ is approximated by Scilab as the value returned by $4 * atan(1.0)$,
- the *sin* function is approximated by a polynomial.

This article is too short to make a complete presentation of the computation of elementary functions. The interested reader may consider the direct analysis of the Fdlibm library as very instructive [24]. The "Elementary Functions" book by Muller [16] is a complete reference on this subject.

In Scilab, the "sin" function is directly performed by a fortran source code (sci_f_sin.f) and no additionnal algorithm is performed directly by Scilab. At the compiler level, though, the "sin" function is provided by a library which is compiler-dependent. The main structure of the algorithm which computes "in" is probably the following

- scale the input x so that in lies in a restricted interval,
- use a polynomial approximation of the local behaviour of "sin" in the neighbourhood of 0, with a guaranteed precision.

In the Fdlibm library for example, the scaling interval is $[-\pi/4, \pi/4]$. The polynomial approximation of the sine function has the general form

$$sin(x) \approx x + a_3 x^3 + \ldots + a_{2n+1} x^{2n+1} \quad (100)$$
$$\approx x + x^3 p(x^2) \quad (101)$$

In the Fdlibm library, 6 terms are used.

For the inverse tan "atan" function, which is used to compute an approximated value of π, the process is the same. All these operations are guaranteed with some precision. For example, suppose that the functions are guaranteed with 14 significant digits. That means that 17-14 + 1 = 3 digits may be rounded in the process. In our current example, the value of $sin(\pi)$ is approximated with 17 digits after the point as "0.00000000000000012". That means that 2 digits have been rounded.

A.3 One more step

In fact, it is possible to reduce the number of significant digits of the sine function to as low as 0 significant digits. The mathematical theory is $sin(2^n\pi) = 0$, but that is not true with floating point numbers. In the following Scilab session, we

```
-->for i = 1:5
-->k=10*i;
-->n = 2^k;
-->sin(n*%pi)
-->end
 ans  =
  - 0.00000000000001254038322
 ans  =
  - 0.000000000012841352420 63
```

```
ans  =
 - 0.000000131495448 7872237
ans  =
 - 0.000134651339151 2239052
ans  =
 - 0.137446488227798 5633419
```

For $sin(2^{50})$, all significant digits are lost. This computation may sound *extreme*, but it must be noticed that it is inside the double precision possibility, since $2^{50} \approx 3.10^{15} \ll 10^{308}$. The solution may be to use multiple precision numbers, such as in the Gnu Multiple Precision system.

If you know a better algorithm, based on double precision only, which allows to compute accurately such kind of values, the Scilab team will surely be interested to hear from you !

References

[1] Loss of significance. http://en.wikipedia.org/wiki/Loss_of_significance.

[2] Quadratic equation. http://en.wikipedia.org/wiki/Quadratic_equation.

[3] Quadratic equation. http://mathworld.wolfram.com/QuadraticEquation.html.

[4] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. 1972.

[5] J.T. Coonen. Underflow and the denormalized numbers. *Computer*, 14(3):75–87, March 1981.

[6] George E. Forsythe. *How Do You Solve A Quadratic Equation ?* Computer Science Department, School of Humanities and Sciences, Stanford University, JUNE 1966. ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/66/40/CS-TR-66-40.pdf.

[7] George E. Forsythe. How do you solve a quadratic equation ? 1966. ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/66/40/CS-TR-66-40.pdf.

[8] Paul Friedland. Algorithm 312: Absolute value and square root of a complex number. *Commun. ACM*, 10(10):665, 1967.

[9] David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. Association for Computing Machinery, Inc., March 1991. http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf.

[10] P. Dugac J. Dixmier. *Cours de Mathématiques du premier cycle, 1ère année*. Gauthier-Villars, 1969.

[11] M. A. Jenkins. Algorithm 493: Zeros of a real polynomial [c2]. *ACM Trans. Math. Softw.*, 1(2):178–189, 1975.

[12] W. KAHAN. Branch cuts for complex elementary functmns, or much ado about nothing's sign bit. pages 165–211, 1987.

[13] W. Kahan. On the cost of floating-point computation without extra-precise arithmetic. 2004. http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf.

[14] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Third Edition, Addison Wesley, Reading, MA, 1998.

[15] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision blas. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.

[16] Jean-Michel Muller. *Elementary functions: algorithms and implementation*. Birkhauser Boston, Inc., Secaucus, NJ, USA, 1997.

[17] Yves Nievergelt. How (not) to solve quadratic equations. *The College Mathematics Journal*, 34(2):90–104, 2003.

[18] W. H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, Second Edition*. 1992.

[19] Douglas M. Priest. Efficient scaling for complex division. *ACM Trans. Math. Softw.*, 30(4):389–401, 2004.

[20] K.E. Schmidt. Numerical derivatives. http://fermi.la.asu.edu/PHY531/intro/node1.html.

[21] Robert L. Smith. Algorithm 116: Complex division. *Commun. ACM*, 5(8):435, 1962.

[22] Robert L. Smith. Algorithm 116: Complex division. *Commun. ACM*, 5(8):435, 1962.

[23] G. W. Stewart. A note on complex division. *ACM Trans. Math. Softw.*, 11(3):238–241, 1985.

[24] Inc. Sun Microsystems. A freely distributable c math library. 1993. http://www.netlib.org/fdlibm.

[25] P. Wynn. An arsenal of ALGOL procedures for complex arithmetic. 2(4):232–255, December 1962.